

|  |
| --- |
| **­­­­­­­­­­­­­FACULDADE DE TECNOLOGIA**  **DE SÃO CAETANO DO SUL** |

**Vinícius Temóteo Ferrari**

**Test-Driven Development como alternativa de garantia de qualidade no desenvolvimento de software orientado a objeto**

São Caetano do Sul/SP  
2013

**Vinícius Temóteo Ferrari**

Test-Driven Development como alternativa de garantia de qualidade no desenvolvimento de software orientado a objeto

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia de São Caetano do Sul, sob a orientação do Professor Msc. Wilson Vendramel, como requisito parcial para a obtenção do diploma de Graduação no Curso de Analise e Desenvolvimento de Sistemas

São Caetano do Sul/SP  
2013

**Lista de figuras**

**Lista de abreviaturas e siglas**

TDD - Test-driven development

TF – Test First

UML - Unified Modeling Language

XP – eXtreme Programing

Sumário

[1.1. Objetivos 7](#_Toc415439568)

[1.2. Justificativa 7](#_Toc415439569)

[1.3. Materiais e procedimentos 9](#_Toc415439570)

[2. FUNDAMENTAÇÃO TEÓRICA 11](#_Toc415439571)

[2.1. Qualidade 11](#_Toc415439572)

[2.2. Qualidade de produto de software 17](#_Toc415439573)

[2.3. Qualidade de código 17](#_Toc415439574)

[2.3.1. Complexidade Ciclomática (CC) 18](#_Toc415439575)

[2.3.2. Número de Linhas De Código (LOC – Line of Code) 19](#_Toc415439576)

[2.3.3. Dívida Técnica (Technical Debt) 19](#_Toc415439577)

[2.3.4. Cobertura de código. 20](#_Toc415439578)

[2.3.5. Duplicação de código. 20](#_Toc415439579)

[3. TDD – Alternativa de garantia de qualidade no desenvolvimento de software 22](#_Toc415439580)

[3.1. JUnit: Framework para testes automáticos 22](#_Toc415439581)

[3.2. Passos do TDD 23](#_Toc415439582)

[3.2.1. Vermelho: Criando um novo teste 23](#_Toc415439583)

[3.2.2. Verde: Fazendo o teste passar 25](#_Toc415439584)

[3.2.3. Refatorar: Generalizar o método 26](#_Toc415439585)

[3.2.4. Teste 27](#_Toc415439586)

[4. Ferramenta de publicação de Livro Digital no formato e-Pub: implementação utilizando os conceitos de TDD 29](#_Toc415439587)

[4.1. Configuração do projeto 29](#_Toc415439588)

[4.2. Ferramentas utilizadas 29](#_Toc415439589)

[4.3. Demonstração do TDD no desenvolvimento do modulo 29](#_Toc415439590)

[4.3.1. Criar listas de apoio 30](#_Toc415439591)

[4.3.2. Exemplo de código mínimo de teste etapa 1 30](#_Toc415439592)

[4.3.3. Implementação do código 32](#_Toc415439593)

[4.3.4. Refatoração 33](#_Toc415439594)

[4.3.5. Modelo de UML 33](#_Toc415439595)

[5. Resultados Obtidos 35](#_Toc415439596)

[6. Bibliografia 36](#_Toc415439597)

1. INTRODUÇÃO

O desenvolvimento de software passou por incríveis mudanças desde o início da era da informática, o avanço de linguagens e formas de como trabalhar a favor das linguagens e as novas metodologias, trouxeram o reuso e reaproveitamento de linhas de código, transformou a linguagem estruturada, que era difícil de compreender, e também dar suporte a sistemas antigos ou sistemas de outros programadores; trazendo ao mundo real o modelo de objetos.

Os sistemas orientados a objetos deveriam ser mais fáceis de entender, ler, dar manutenção, porém, problemas parecidos com o modelo estruturado tornaram a aparecer como: códigos extensos; inconsistentes; duplicado. Obrigaram especialistas a criar maneiras de tornar os sistemas padrões mais fáceis de modificar e compreender, criando linguagens de modelagem como a UML – *Unified Modeling Language* – e até ferramentas para melhorar a qualidade no decorrer do desenvolvimento do software. Mesmo assim o tempo, recursos e custos despendidos em direção a qualidade ainda é grande, e há o risco de no final o projeto já se tornar obsoleto.

Os estudiosos de engenharia de software buscam diminuir o risco tanto da obsolescência quanto dos altos custos e tempo. Pensando nisso Beck (2010) defende o Desenvolvimento Guiado por Testes do inglês Test-Driven Developement (TDD).

Resumidamente, o TDD guia através de pequenos passos para que se alcance o Todo, o objetivo, que é a conclusão do projeto, baseando-se em especificações que foram previamente definidas, dessa forma caso precise fazer qualquer mudança basta rodar os testes e assim garantir que o Todo continue a funcionar.

TDD não é somente isso, tem muito mais a agregar e a colaborar com a engenharia de software, entretanto, o TDD é recente e mesmo profissionais com experiência ainda não o conhece, além de haver alguma resistência justamente pela falta de conhecimento e de toda sua potencialidade.

Kent Beck, considerado o criador do TDD, lançou a obra TDD Desenvolvimento Guiado por Testes, o qual apresenta um conjunto de técnicas de linguagem acessível para qualquer engenheiro de software, com a intenção de encorajar a aplicabilidade de TDD em qualquer projeto e assim proporcionar oportunidades para que os profissionais do ramo possam trabalhar mais próximo do real potencial.

No caso desta obra, o mesmo aplicou prioritariamente na área financeira, e apresenta exemplo com foco neste ramo.

Com base na importância e colaboração que o TDD pode acrescentar e agregar à engenharia de software, através desta pesquisa o pesquisador não busca esgotar o assunto, mas trazer à luz da pesquisa científica através desta pesquisa a exposição de uma aplicabilidade de TDD por meio de um exemplo prático.

Para tanto, o pesquisador adota a metodologia da pesquisa quantitativa, guiando-se por autores, através de livros, artigos idôneos, e utiliza imagens e vídeos para compor a apresentação teórica e prática de sua pesquisa acadêmi

O projeto tem por objetivo desenvolver um produto web utilizando Test-Driven Development (TDD) como alternativa de garantia de qualidade no desenvolvimento de software de forma a estudar sobre a qualidade na produção de software;O projeto deve ser capaz de interpretar texto sinalizado com caracteres especias para marcar e transformar em um arquivo e-Pub (e-book, livro digital), além de estudar as métricas de qualidade na produção da aplicação, garantir a legibilidade do código produzido, analisar a escalabilidade e a tolerância à mudança de requisitos do sistema, e também a influencia do TDD no design do projeto.

Trata-se de algo complexo e delicado o desenvolvimento de um software, mesmo compondo uma significante parcela do nosso dia-a-dia. Profissionais de todos os ramos munem-se de softwares para facilitar tarefas difíceis, ou repetitivas atividades profissionais. À medida que a complexidade do problema aumenta a qualidade dele tende a diminuir, isso porque é difícil documentar todas as necessidades, com isso houve uma revolução nos métodos de desenvolvimento surgindo o manifesto ágil, que aconteceu em 2001. Foram definidos vários padrões tanto de modelagem quanto de desenvolvimento ou até métodos de melhores práticas para tentar solucionar os principais problemas de qualidade de software e entre produtos de softwares que satisfazem os *stakeholders*. O trecho do artigo abaixo mostra que durante um processo de desenvolvimento de software a qualidade e a satisfação dos *stakeholders*, foram garantidas ao utilizar o desenvolvimento ágil somado a técnica de TDD.

*Of course, defect rate is just one small measure of quality. If you could talk to our business stakeholders, you’d find they’re satisfied, often even delighted, that we deliver just what they asked for each iteration. We’ve implemented features that competing companies thought were too complex to automate. Many factors come into play, including other agile practices such as short iterations, continuous integration, and refactoring. However, I think our ability to understand and capture business requirements, and then code to meet them, is a critical reason for our success.* (CRISPIN, 2006, p. 71, *grifo do autor*)

Claro, a taxa de defeito é apenas uma pequena medida de qualidade. Se você pudesse falar com nossos stakeholders do negócio, você acharia que eles estão satisfeitos, muitas vezes até mesmo prazer, que nós entregamos exatamente o que eles pediram para cada iteração. Nós implementamos características que as empresas concorrentes pensavam eram demasiado complexas para automatizar. Muitos fatores entram em jogo, incluindo outras práticas ágeis como iterações curtas, integração contínua e refatoração. No entanto, eu acho que a nossa capacidade de compreender e capturar os requisitos de negócio e, em seguida, o código para encontrá-los, é uma razão fundamental para o nosso sucesso. (Tradução nossa).

Após o manifesto ágil, métodos de desenvolvimento foram criados como *Scrum* e *eXtreme Programing,* eles tentam estabelecer parâmetros de qualidade e permitir que o desenvolvimento seja versátil e fácil de ser modificado, e que são passiveis a mudança de requisitos, partindo do XP o TDD é só um pedaço do *framework* e por esse motivo pode ser adaptado para qualquer modelo de desenvolvimento de software.

Escrever código que possa ser compreendido por outro desenvolvedor ou por uma equipe de desenvolvimento é um desafio para engenharia de software e o TDD tenta solucionar o problema obrigando cada unidade de código, cada método ou função, seja testada antes de ser alterada, impedindo falhas que podem ser ocasionadas durante a refatoração.

*Supondo por um momento que tal estilo de programação é possível, ainda pode-se reduzir significativamente a densidade de defeitos de código e fazer o tema de trabalho claro como cristal para todos os envolvidos. Se assim for, então escrever apenas aquele código que é requerido por testes que falham também tem implicações sociais.* (BECK, 2010, p. x).

O TDD pode também diminuir o tempo de desenvolvimento do software, bem como dos testes, pois o tempo de depuração do código diminui devido à técnica aplicada e exigida, por quanto os passos de implementação sejam pequenos.

*If you like numbers, many studies have shown that TDD does improve quality. According to David Janzen’s paper, “Software Architecture Improvement through Test-Driven Development” (ACM Press, 2005), industry studies show that programmers using TDD produced code that passed between 18 and 50 percent more external test cases than code produced by control groups not using TDD. Additionally, the TDD programmers spent less time debugging. Janzen also cites academic studies that demonstrated significant improvements in external software quality and programmer productivity.* (CRISPIN, 2006, p. 71, *grifo do autor*)

Se você gosta de números, muitos estudos têm mostrado que TDD faz melhorar a qualidade. De acordo com o artigo de David Janzen, "Software Improvement Arquitetura através de Test-Driven Development" (ACM Press, 2005), os estudos da indústria mostram que os programadores que usam TDD produzido código que passou entre 18 e 50 por cento de casos de teste mais externos do que código produzido por grupos de controle não usando TDD. Além disso, os programadores TDD gastou menos tempo de depuração. Janzen também cita estudos acadêmicos que demonstraram melhorias significativas na qualidade de software externo e produtividade do programador. (Tradução nossa)

Durante a programação os desenvolvedores sentem-se mais seguros em seguir adiante, pois os passos anteriores estão cobertos por teste. A criação de novos códigos associados fica mais eficaz, devido aos testes de unidade garantir que caso alguma alteração quebre as associações entre os métodos ou classes, os testes, informarão rapidamente o desenvolvedor ou o time que esta desenvolvendo o sistema de software, permitindo rápida correção.

O TDD obriga o desenvolvedor a atender as necessidades do produto diminuindo a quantidade de implementações desnecessárias, mantendo a concentração na necessidade real daquele problema, em um software, ajuda o programador a escrever só o necessário para prosseguir para próxima necessidade do sistema de software. Com testes já estabelecidos pelo programador, qualquer alteração no sistema que gere falha, é imediatamente capturada pela ferramenta de teste garantindo que o software funcione completamente na entrega seja do produto final, módulo ou de um protótipo do sistema.

* 1. Materiais e procedimentos

Para a pesquisa serão estudadas algumas literaturas sobre o assunto, serão utilizados recursos, tanto impressos (livros), como digitais (e-livros) encontrados em bibliotecas ou na internet. Imagens e vídeos também serão analisados para compor o estudo.

O projeto tem por objetivo desenvolvimento de um, módulos ou protótipos de sistemas de software capaz de gerar um arquivo no formato e-Pub, utilizando tal técnica de programação. O software utiliza ferramentas livres de código aberto, como: maven – ferramenta de build e dependencia de condigos; netbeans – IDE para desenvolvimento de codigo; Java Server Faces (JSF) – framework web e também, é utilizado um framework para testes (JUnit), próprio para linguagem Java. O sistema foi desenvolvido na mesma linguagem de programação traz exemplos com o código escrito, guiado pelo TDD.

1. FUNDAMENTAÇÃO TEÓRICA

Texto explicativo do capitulo

* 1. Qualidade

Weinberg (1993) aponta a questão da relatividade quanto à qualidade, para ele é preciso examinar com muito cuidado porque definições como “a qualidade é a conformidade às exigências” se transformam em “às exigências de alguma pessoa” isto porque o mesmo produto detém aspectos de qualidade diferentes de acordo com cada ponto de vista. Assim, Weinberg (1993, p. 6) quanto às definições de qualidade de software ele se pergunta “Quem é a pessoa por trás dessa definição de qualidade?”

Weinberg (1993, p. 6) cita algumas “ideias familiares”, porém conflitantes sobre qualidade de software:

*Defeito zero é a alta qualidade*

a) Para os usuários cujo trabalho é afetado pelos defeitos.

b) Para os gerentes que são criticados pelos defeitos.

*Ter um grande número de funções é alta qualidade*

a) Para os usuários cujo trabalho pode tirar proveito dessas funções – se eles as conhecerem.

b) Para os distribuidores que acreditam que as funções vendem produtos.

*Codificação elegante é alta qualidade*

a) Para o pessoal de desenvolvimento que dá um grande valor às opiniões de seus colegas.

b) Para os professores de ciência da computação que apreciam elegância.

*Alto desempenho é alta qualidade*

a) Para os usuários cujo trabalho sobrecarrega a capacidade de suas máquinas.

b) Para o pessoal de venda que tem de submeter seus produtos à benchmarks.

*Baixo custo de desenvolvimento é alta qualidade*

a) Para os usuários cujo trabalho sobrecarrega a capacidade de suas máquinas.

b) Para os gerentes do projeto que estão com orçamentos apertados.

*Desenvolvimento rápido é alta qualidade*

a) Para os usuários cujo trabalho está esperando pelo software.

b) Para os distribuidores que desejam colonizar um mercado antes de seus concorrentes.

*Facilidade para o usuário (userfriendliness) é alta qualidade*

a) Para os usuários que gastam oito horas por ida na frente de uma tela utilizando o software.

b) Para os usuários que não conseguem se lembrar de detalhes de interface (WEINBERG, 1993, p. 6-7).

Weinberg (1993) ressalta aspectos importantes sobre qualidade por meio da conhecida frase “Mais qualidade para uma pessoa pode significar menos qualidade para outra”, e explica que se o objetivo de um projeto for Qualidade Total a opção a partir da definição citada, seria a de desenvolver um processo que abrangesse todos as pessoas relevantes, atribuir uma medida de qualidade para cada um, e a soma dessas medidas resultaria na qualidade total para cada abordagem diferente.

No entanto, o autor diz que “na prática, é claro, nenhum projeto de desenvolvimento de software utiliza sempre tal processo elaborado”, para ele isso não ocorre porque as maiorias das pessoas são previamente eliminadas por quem toma a decisão, o que o autor faz uma reflexão “Qual opinião sobre qualidade deve ser levada em conta quando da tomada de decisão?”

Para Weinberg (1993, p. 8) a tomada de decisão nem é um negócio racional, pois geralmente a decisão tem uma base emocional, daí a “definição de qualidade é sempre política e emocional”, e se justifica exemplificando que a decisão de um gerente de projeto sobre a qualidade de um processador de texto pode ser diferente se alguém de seu círculo emocional (familiar etc) estiver envolvido.

Porque para Weinberg (1993, p. 8) “ela sempre envolve uma série de decisões sobre quais opiniões contam e quanto elas pesam umas em relação às outras”, porém estas decisões usualmente ficam “escondidas da visão do público”; e o fazem “[...] escondidas mesmo das mentes conscientes das pessoas que as tomam”.

A isto se soma a questão “valor” o quanto alguém esta disposto a fazer (ou pagar) para que suas necessidades, requisitos sejam alcançados. Weinberg (1993) relata um episódio pessoal, que consistia na lembrança de seu pai jogando com ele Cribbage (do poeta sir John Suckling), um jogo pouco popular em algumas regiões o que dificultava encontrar parceiros para jogar. Entretanto, ficou muito feliz ao descobrir o programa de cribbage chamado Precision Cribbage (Macintosh), o qual mesmo descobrindo dois escores errados no jogo ele ficou suficientemente satisfeito a ponto de jogar por horas e ainda pagar a taxa voluntária de distribuição.

Com este exemplo o autor elucida a questão valor, o qual detém sua importância individual.

Além disso, há outros aspectos sobre qualidade, no caso, sobre padrões de opinião, com o tempo não se busca melhorar se determinado software já é superior que a concorrência, mesmo reconhecendo algum ou alguns erros no programa. Assim, se há pessoas que utilizam o produto e estão satisfeitas é decidido que tem boa qualidade; quanto aos erros, “todos os softwares têm erros (no mínimo, nós não podemos provar o contrário)”; e se há pessoas que o escolhe ao invés de comprar o da concorrência é porque detém qualidade superior (WEINBERG, 1993, p. 11).

Por isso que há pouca motivação para melhorar a qualidade, exceto quando fatores como esses se reverterem, daí poderão servir de motivação para a busca por melhoria na qualidade dos mesmos; ou seja, sairão da estagnação, até que sintam satisfeitos com o que foi produzido, e aí novamente a estagnação.

Entretanto, quando se trata de programa há outro fator que também contribui para a estagnação, ou simplesmente, porque há tanta desistência em melhorar algum programa: “Como reduzir defeitos de software para menos que um em um milhão de linhas de código?” (WEINBERG, 1993, p. 12).

Crosby (apud WEINBERG, 1993, p. 12) sobre melhorias em um programa, usualmente iniciam com um estudo do custo da qualidade (valor da qualidade), e geralmente os gerentes mantém o pensamento em cortar custos ou acelerar um cronograma, porém quanto ao “[...] valor da qualidade melhorada parece ser algo que eles nunca pensaram em medir”.

Weinberg (1993, p. 13) explica que “na medida em que uma organização começa a entender o real valor da qualidade, sua motivação para melhorar aumenta” e isso ocorre porque a partir daí isso o guiará a ter um melhor entendimento de como melhorar e quase consequentemente, o “[...] levará a um melhor entendimento do valor da qualidade”.

Existe a necessidade do software passar pelo teste do software, que consiste em uma investigação por meio de um processo da engenharia de software que envolve ações a fim de testar o bom funcionamento e a qualidade do mesmo.

Esse processo é realizado pelo testador de software que aplica ações que vão desde o levantamento de requisitos até a execução do teste propriamente dito, visto que o projeto a ser desenvolvido muitas vezes demanda muitas pessoas envolvidas, e muitas vezes possuem um volume significativo de estados, com fórmulas, atividades e algoritmos complexos, e todo esse universo o torna mais complexo, e mais difícil de garantir que o software funcione corretamente sem erros, daí a necessidade do testador de software.

Mas independentemente da metodologia de trabalho adotada, já visando obter o melhor produto final, tornou-se imprescindível a busca constante pela melhoria dos processos de engenharia de software. Assim, com o tempo tomou-se por hábito utilizar como base modelos sugeridos por entidades internacionais balizadas, como os conhecidos os SW-CMM, SE-CMM, ISO/IEC 15504 e o mais conhecido CMMI.

Mesmo assim, os testes de softwares mesmo com os riscos e falhas de teste, ainda são vitais e indispensáveis, contudo recai em outra problemática, o testador de software, se for ineficiente ou agir de maneira subestimada, até por falta de tempo, recursos humanos ou financeiros, a etapa imprescindível pode não ser a resposta à garantia de qualidade do software.

Para Weinberg (1993, p. 1) sobre qualidade, “produzir e manter software não constitui uma série aleatória de eventos. Existem padrões, e esses padrões oferecem uma oportunidade de controlar nossos produtos, nossas organizações e nossas vidas”;

Brooks Jr. (2009) já disse que gerenciar um projeto de programação de computadores é como gerenciar qualquer outro empreendimento, porém o saber é cumulativo, e lida-se com uma arte que demanda muita criatividade, especialmente se baseada em boas técnicas.

Mas nem tudo são alegrias, há as tristezas, como ter de produzir perfeitamente, são os outros que definem os objetivos, fornecem os recursos e trazem a informação, a grande dependência de programas que foram desenvolvidos por outros profissionais que tendem a ser mal projetados, ou não tem código-fonte ou com documentação incompleta, a partir daí o programador tem que investir muitas horas corrigindo o que na teoria deveria estar pronto, disponível e utilizável; os longos momentos de horas melancólicas de trabalho monótono e cansativo (BROOKS JR, 2009, p. 8).

A pior das tristezas é a que as vezes, o produto que se investiu tanto tempo, esforços, recursos, parece obsoleto quando fica pronto, e seus concorrentes já estão buscando novas e melhores ideias (BROOK JR, 2009, p. 8).

Pensando muito mais nos contra que nos prós, muita pesquisa e estudos tem sido feitas em busca de encontrar meios para transformar parte desse processo em algo mais agradável e seguro, e especialmente diminuir as chances de se investir tempo e recursos em algo que pode ficar obsoleto, e se possível reduzir consideravelmente o tempo de produção e erros.

“Clean Code de Roberto C. Martin e descreve os princípios necessários para um código legível, elegante e eficiente”.

Um código limpo é simples e direto. [...] tão bem legível quanto uma prosa bem escrita. [...] jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas (MARTIN apud BOOCH, 2011,p. 3)

Além de seu criador, um desenvolvedor pode ler e melhorar um código limpo. Ele tem testes de unidade e de aceitação, nomes significativos; ele oferece apenas uma maneira, e não várias, de se fazer uma tarefa; possui poucas dependências, as quais são explicitamente declaradas e oferecem uma API mínima e clara. O código deve ser inteligível já que dependendo da linguagem, nem toda informação necessária pode expressar no código em si (MARTIN apud THOMAS, 2011,p. 4)

Por que um código limpo?

Segundo Beck (2010, prefácio) há muitas razões para se investir em um código limpo, por ser uma forma previsível de desenvolver; sabe-se exatamente quando acabou; a certeza de que não ficou para trás uma trilha de erros; maior oportunidade de aprender com cada código fonte; melhora-o para o usuário e consequentemente suas vidas; melhor ambiente de trabalho; mais satisfação em escrevê-lo. Ao se reduzir a densidade de defeitos então a garantia da qualidade pode mudar de trabalho reativo para pró-ativo, e muitas outras boas razões.

Mas como obter um código limpo que funcione?

“Conduzindo o desenvolvimento com testes automatizados, um estilo de desenvolvimento chamado Desenvolvimento Guiado por Testes” (TDD) (2010, prefácio).

Beck xxx, p. prefácio alerta que através desse o TDD, escreve-se o código novo apenas se um teste automatizado falhar; e eliminando duplicação.

Seguindo essas duas regras, Beck diz que isso ira gerar “um complexo comportamento individual e de grupo com implicações técnicas”, deve-se:

“[...] projetar organicamente com código, executando e fornecendo feedback entre as decisões; escrever seus próprios testes; o ambiente de desenvolvimento deve fornecer resposta rápida a pequenas mudanças; o projeto deve consistir em muitos componentes altamente coesos e fracamente acoplados para tornar os testes fáceis” (BECK, 2010, p. prefácio).

Assim se organizou com o esquema ‘vermelho/verde/refator’ que consiste em vermelho ‘escrever um pequeno teste que não funcione e que talvez nem compile inicialmente; verde ‘fazer rapidamente o teste funcionar, mesmo cometendo algum pecado necessário no processo ; e refatorar significa eliminar todas as duplicadas criadas apenas para que o teste funcione”.

Mas apesar das inúmeras vantagens apontadas, pode ocorrer alguma resistência, engenheiros de software pode se perguntar por que escrever testes automatizados e assim dar passos tão pequenos, para Beck (2010, p. xi) o que torna possível, primeiramente, é a coragem.

Feathers (p. 86) diz que TDD é o recurso mais valioso e é a mais poderosa tecnica de inclusão de recursos, […] . Ela permite consentrarmos em uma caracteristica de cada vez. (p 91)

Beck (2010, p. prefácio) explica que o medo convive com o programador e o faz hesitar, ele se comunica menos, o afasta do *feedback*, torna-o mal humorado, e o TDD possibilita transformar positivamente, ou seja, facilita inverter cada uma dessas situações.

O autor explica que através do TDD quando um teste esta em funcionamento, ele esta em funcionamento naquele momento e no futuro, e assim mais próximos de ter tudo funcionando, mas para tanto, ele sugere que “quanto mais árduo for o problema de programação menor deve ser o terreno que cada teste deve cobrir”.

* 1. Qualidade de produto de software

Existe atualmente uma área totalmente voltada para garantir qualidade em produtos de software, órgãos como ISO, ABNT, entre outros, criam norma e especificam padrões de qualidade, atingir tais padrões é o desafio da engenharia de software desde o início das linguagens de programação.

Definir qualidade é um assunto tão difícil quanto tentar chegar nela, especialistas do mundo todo debatem formas de conseguir atingir todos os pontos de qualidade, desde o código, até o suporte do mesmo. Modelos de negócio, gestão de áreas, tudo isso entra em conta, quando se quer medir qualidade, porem quando pensamos exatamente o que é necessário e o que é preciso, conseguimos estabelecer um parâmetro e então avaliar em qual nível o produto esta.

* 1. Qualidade de código

Martin Fowler desse: “Qualquer tolo consegue escrever código que um computador entenda. Bons programadores conseguem escrever código que humanos conseguem entender” (p 15, Refactoring: Improving the design of existing Code); essa frase é muito citada no mundo agile e muitos desenvolvedores tentam adotar, porem mesmo em seu livro, Fowler diz que se você precisa refatorar é essencial ter uma bateria solida de testes(p,89), já que isso evita problemas de mudança de comportamento entre as refatorações.

Outro problema comum de código mal escrito é que as pessoas mais antigas da equipe que conhecem o código, e são muitas vezes impedidos de afastarem desse código, já que são os principais conhecedores dos mesmos, e quando uma nova funcionalidade difícil é pedida essa pessoa que é encarregada a introduzir o código no sistema. Esse cenário é comum em empresas de todo o mundo. (apostila caelum, p 51).

E para tentar assegurar qualidade foram criadas métricas de qualidade de código.

* + 1. Complexidade Ciclomática (CC)

Esta métrica de qualidade de software foi formulada por Thomas J. McCabe em 1976, (p1) e mede a quantidade de caminhos possíveis que um método, uma classe ou arquivo, contém e é calculado através de um grafo direcionado.

Em um de seus artigos ele questiona a necessidade forma de medir a facilidade de manter e testar um método; ele questiona o fato de os softwares nessa época gastar valores de dinheiro consideráveis em manutenção e teste;

Para isso McCabe (p1) desenvolveu um teorema matemático guiado nos caminhos possíveis de um código; e quanto maior a quantidade de caminho, moires as confeiçoes a serem testadas e cobertas, além de uma necessidade maior de entendimento.

O teorema de forma resumida é a quantidade de setas do grafo representado pela letra e, menos a quantidade de nós que o algoritmo tem representado por n, mais a quantidade de componentes conectados representado por p, que resulta na complexidade ciclo matica v(G).

v(G) = e – n + p

Ou seja, o método da figura N

http://www.devmedia.com.br/sequencia-de-fibonacci-em-java/23620

Tem uma Complexidade Ciclomática de **x** dada a formula v(G) = 7 – 3 + 1. Já o mesmo programa a da figura (N+1) tem a Complexidade Ciclomática de **y** dada a formula v(G) = e – n + p.

* + 1. Número de Linhas De Código (LOC – Line of Code)

São as Linhas de Códigos efetivas em um sistema.

Considerada a medida mais simples e também a menos precisa, já que não existem distinção de linhas em branco, comentários, e código, a métrica LOC serve como referência para identificar o tamanho do projeto, pois, um programa de 4 milhões de linhas tende a ser mais complexo que um programa de 4 mil linhas. Koscianski (p,229, qualidade de software, 2ª ed) .

Em seu exemplo Koscianski mostra que a figura X.X tem três linhas, sendo três logicas e duas físicas

// uma comparacao

if (a > b) { a++; puts (“ok”);

}

* + 1. Dívida Técnica (Technical Debt)

O conceito de dívida técnica surgiu com através de uma metáfora feita por Ward Cunningham,

*Nesta metáfora, fazendo as coisas da maneira rápida e suja nos leva à uma dívida técnica, que é semelhante a uma dívida financeira. Como uma dívida financeira, a dívida técnica incorre em pagamentos de juros, que vêm sob a forma de o esforço extra que temos de fazer no futuro desenvolvimento devido à escolha rápida e suja de design.*

(Fowler, <http://martinfowler.com/bliki/TechnicalDebt.html>, 2003)

Pouco de pois que Cunningham fez essa metáfora a comunidade de desenvolvimento de software passou a evoluir o conceito, considerando a dívida técnica um dos principais aspectos de qualidade, já que um software com baixa dívida técnica tende a ser menos difícil de entender, mais seguro de modificar, exige menor tempo de manutenção, que consequentemente diminui o custo do projeto.

(agile Desenvolvimento de software com entregas frequentes e foco no valor de negocio - Andŕe Faria Gomes, p 88)

* + 1. Cobertura de código.

Esta métrica é usada para medir a quantidade de código que foi testada ou deixou de ser, normalmente expressa em porcentagem, ou em linhas de código logico.

Uma cobertura de código de 100% indica que todo código de produção tem ao menos um teste passando por ele. (Mauricio Aniche, Test Driven Development - Teste e Design no Mundo Real com DotNet - Casa do Codigo, p139)

* + 1. Duplicação de código.

É o termo usado para o código que foi repetido em mais de um lugar, ele pode ser o mesmo código de outro trecho, pode ser um código parecido mais com uma variável diferente, no entanto o código duplicado é considerado por Fowler, o principal problema de um software, e se isso ocorre é essencial encontrar uma maneira de unificá-los, pois, diminui assim a complexidade do sistema, e sua divida técnica.

Uma maneira de remover código duplicado é introduzir um método ou uma classe, e passar suas dependências por parâmetro.

Por exemplo o código abaixo pode ser representado por um único método e chamado duas vezes

**extern** int array1[];

**extern** int array2[];

int sum1 = 0;

int sum2 = 0;

int average1 = 0;

int average2 = 0;

for (int i = 0; i < 4; i++)

{

sum1 += array1[i];

}

average1 = sum1/4;

for (int i = 0; i < 4; i++)

{

sum2 += array2[i];

}

average2 = sum2/4;

refatorado o codigo fica:

int calcAverage (int\* Array\_of\_4)

{

int sum = 0;

for (int i = 0; i < 4; i++)

{

sum += Array\_of\_4[i];

}

return sum/4;

}

**extern** int array1[];

**extern** int array2[];

int average1 = calcAverage(array1);

int average2 = calcAverage(array2);

(Wikipedia)

1. TDD – Alternativa de garantia de qualidade no desenvolvimento de software

TDD é uma técnica de programação criada por um membro do manifesto ágil Kent Beck, que extraiu do modelo de desenvolvimento ágil XP, também criado por ele. A técnica é simples e muitas vezes repetitiva já que são necessários somente três passos para começar a aplicar a prática, Beck (2010, p. prefácio) define os passos como sendo:

1. Vermelho – Escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente.
2. Verde – Fazer rapidamente o teste funcionar, mesmo contendo algum pecado necessário inicialmente.
3. Refatorar – Eliminar todas as duplicatas criadas apenas para que o teste funcione. (BECK, 2010).

No meio informal da informática é considerado um mantra tais passos para a programação, a Figura 1 mostra a relação das palavras citadas por Beck com o objetivo de cada passo.

Figura 1: Mantra do TDD



Fonte Diogoosorio, 2012

* 1. JUnit: Framework para testes automáticos

A principal ferramenta para se desenvolver em TDD é o JUnit uma ferramenta para automatizar testes de unidade em softwares desenvolvidos em Java, esta ferramenta permite a execução de grandes quantidades de testes através de modo textual ou modo gráfico.

Massol (2005) tenta descrever de maneira simples o que é um framework e como o JUnit se enquadra em um framework.

Um framework é um aplicativo semicompleto. Um framework fornece uma estrutura reutilizável e comum que pode ser compartilhada entre aplicativos. Os desenvolvedores incorporam o framework em seu próprio aplicativo e o estendem até que eles cumpram suas necessidades especificas. Frameworks diferem de kits de ferramentas porque eles fornecem uma estrutura coerente, ao invés de um conjunto de classes utilitárias.

Framework, segundo a definição de alguns autores, refere-se a um conjunto de classes que constitui um projeto abstrato voltado à solução de uma família de problemas; com uma arquitetura definida e desenvolvida para uma família de subsistemas, o qual oferta os construtores básicos para criá-los, e para atingir a máxima reutilização com grande potencial de especialização; e exposto *hot-spots* (pontos de extensão) onde devem ser feitas as adaptações do código para um funcionamento específico de certos módulos ***(MATTSSON, 1996; MATTSSON, 2000; JOHNSON, 1991; GAMMA et al, 1995; BUSCHMANN et al, 1996; PREE, 1995; PINTO, 2000). Daonde saiu isso???***

Produzir enquanto roda uma bateria de teste é mais uma facilidade que o framework pode proporcionar, ele foi escrito em código aberto e pode ser visualizado todo o comportamento e modificado quando necessário.

Não existem dificuldades de utilização já que pode ser criado pacotes de teste em separado, a implementação de uma interface garante todas as funcionalidades de teste, coso o usuário necessite de mais funcionalidades, existem API’s que podem criar objetos fictícios de conexão com banco ou de acesso a outros serviços, existem também ferramenta para verificar comportamento de requisições HTTPs.

* 1. Passos do TDD

Texto Explicativo.

* + 1. Vermelho: Criando um novo teste

O primeiro passo, o software ainda não existe e deve-se criar o primeiro teste unitário para resolver o problema proposto, aqui, o programador coloca exatamente o que a unidade do programa deve realizar, e depois que especificado, o teste deve-se rodar o framework de teste e esperar o retorno de falha.

As figuras 2 mostra o trecho de código extraído do livro de Beck (2010, p. 17) demonstrando a técnica na pratica, o exemplo que ele propõe é um sistema de gerenciamento de investimento de títulos, que uma companhia vende. Neste exemplo o sistema já existe somente as novas modificações que estão sendo feitas estão utilizando TDD.

Figura 2: Teste não compila



Fonte: Adaptado de Beck, 2010.

A Figura 3 mostra o framework JUnit exibindo os resultados do teste da Figura 1 de forma gráfica, mais a frente será abordado a ferramenta de teste JUnit.

Figura 3: JUnit em modo gráfico junto da IDE NetBeans.



Fonte: Adaptado de Beck, 2010.

* + 1. Verde: Fazendo o teste passar

O próximo passo é resolver o problema dos erros, o framework JUnit vai mostrar que está faltando a classe Dollar, porém ainda existem mais três erros para que o teste passe a funcionar, o correto seria corrigir um erro, rodar os testes, capturar mais um erro e corrigir um de cada vez, entretanto como no exemplo de Beck o programador pode, caso esteja seguro dos próximos passos, ir um pouco além e corrigir alguns erros a mais, contudo nada além da necessidade de passar nos teste, mesmo que implique em métodos retornando valores falsos ou *stubs*.

Neste passo foi implementado o mínimo necessário para que o compilador passe a responder de forma aceitável, resolvendo os erros de compilação e devolvendo pelo menos algum valor de retorno. A figura 4 mostra as linhas adicionadas e o resultado do teste depois da execução.

Figura 4: Resultado dos testes após a implementação de *stubs*.



Fonte: Adaptado de Beck, 2010.

Agora existe um programa compilando, contudo ele não devolve o retorno esperado, quando Beck roda este teste em seu livro, ele afirma que temos progresso no desenvolvimento do software, e sugere a menor implementação possível para que possa fazer o teste passar. Como alteração modifica-se a linha 3 para “public int amount = 10;” obrigando o teste a funcionar, como mostra a figura 5.

Figura 5: forçando o resultado de um teste



Fonte: Adaptado de Beck, 2010.

Com isso pode-se prosseguir para o passo final do ciclo do TDD a refatoração.

* + 1. Refatorar: Generalizar o método

O último passo do “mantra do TDD” é a refatoração, Refatorar o código produzido é um passo delicado, pois é aqui que vamos garantir a qualidade do código ou o fracasso de entendimento do mesmo, em um livro específico de refatoração Martin Fowler, com ajuda de Kent Beck (2004), descrevem as melhores práticas de refatoração, entretanto logo no prefacio Fowler descreve a dificuldade de expor em uma introdução o tema e resolve pular direto para exemplos práticos de refatoração.

O problema do software WyCash consiste em que ainda existe poucas coisas a serem mudadas no novo modulo a ser implementado por isso a refatoração necessária para chegar a um design razoável requer somente mais uma execução do teste. As principais alterações são: remoção da constante 10; implementação do construtor; implementação do método times. A maneira correta de resolver os problemas de refatoração é refatorar gradualmente, porém este exemplo e simples o suficiente para dar passos maiores. A figura 6 mostra o código após a refatoração e o resultado do teste.

Refatoração é uma palavra muito utilizada no campo da informática, Martin Fowler em seu site publicou a definição de refatoração como sendo, “*Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*“, ou na tradução acessada pelo wikipedia “Refatoração é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.”.

Esta técnica tem por objetivo melhorar, além da legibilidade e qualidade no design, como na escalabilidade, desempenho etc. Refatorar código legado é um desafio, muito comum que programadores muitas vezes passam, e com a ausência de código de teste, a chance de fracasso é muito maior. Fowler (2004, p. 82) em seu livro observa dizendo, “Se você quiser refatorar, a pré-condição essencial é ter testes sólidos.”.

Figura 6: Refatoração da classe Dollar e execução do teste.



Fonte: Adaptado de Beck, 2010.

Com esse passo completo, é possível agora voltar ao passo um e prosseguir, adicionando novos testes para estes métodos ou criar um novo teste para ou novas funcionalidades ao sistema.

Mais adiante a pesquisa irá abordar técnicas de refatoração, pois é um dos principais itens que garante a qualidade do software e permite a legibilidade e a compreensão para equipes, além de influenciar na qualidade final do software.

* + 1. Teste

Existem várias formas de teste de sistemas variando de testes funcionais a testes estruturais, ou também conhecidos por testes de caixa preta a testes de caixa branca. Cada ambiente de teste visa garantir a captura de falha em cada etapa de desenvolvimento, os testes podem variar de testes automáticos a testes de usuários.

3.3. Baby-Staps child staps

3.3. Feedbacks dos testes

Fonte Test Driven Development: Teste e design no mundo real com Java

Mauricio Aniche

Figura 7: Efeito *feedback*

1. Ferramenta de publicação de Livro Digital no formato e-Pub: implementação utilizando os conceitos de TDD

Esta etapa do projeto tem por objetivo exemplificar a prática do TDD, durante o processo de desenvolvimento do software, que será aplicado em uma ferramenta de publicação de livros digitais no formato ePub, porém, o processo de transformação em arquivo físico, é parte de uma API de terceiros com código fonte aberto disponível em http://www.siegmann.nl/epublib.

O principal objetivo do projeto é mostrar a criação das páginas em formato texto e transformar em um formato que a API consiga interpretar e transformar no arquivo adequado para a publicação.

* 1. Configuração do projeto

É um projeto na linguagem de programação Java com a utilização de frameworks para desenvolvimento web e uma API de terceiros para publicação no arquivo físico ePub

* 1. Ferramentas utilizadas
* **Netbeans** – Ambiente de desenvolvimento Java distribuído pela Oracle
* **JSF** – Framework web
* **Primefaces** – Framework web
* **Epublib** – Framework ePub
* **JUnit** - – Framework de testes
* **Maven** – Ferramenta de build
* **Git** – Controle de versão
* **Sonar** – Ferramenta de análise de código
  1. Demonstração do TDD no desenvolvimento do modulo

Demostrando como criar uma camada entre a API epub-lib e a camada de usuário, facilitando o processo entre entrada de texto pelo usuário e a transformação em HTML, formatado e especificado sob as regras ePub

O processo de TDD na construção do módulo de transformação de texto em código e publicação do mesmo

* + 1. Criar listas de apoio

Como sugestão para controlar o processo de TDD, é possível fazer a criação de listas para que facilite a evolução dos testes e também o alcance dos objetivos do sistema

Img lista

As listas podem envolver maiores complexidades e podem ser quebradas em sub-listas para que facilite o processo

Imagem

Assim é possível evitar o problema de falta de planejamento e ajuda o início dos testes, assim o sistema não necessita de todo o processo de planejamento inicial, e não deixa o desenvolvedor sem os primeiros passos para o sistema

* + 1. Exemplo de código mínimo de teste etapa 1

Com a lista já minimamente planejada é fácil descobrir qual tarefa é a mais simples, assim é possível começar com o menor passo e menor teste. Os item: transformar \*\* em <b> ; transformar \_\_ em <u>; transformar // em <i>. Aparentam pertencer ao mesmo grupo de código, portanto é possível criar os testes para cada tipo e depois generalizar e remover as duplicações.

Imagem

O primeiro teste mais simples e funcional que pode ser pensado é passar uma String por parâmetro contendo um dos símbolos da lista e esse símbolo devolver o retorno esperado.

@Test

public void transformaAsteriscoEmTagB(){

TransformaTag transforma = new TransformaTag();

assertEquals(“<b>vinicius</b>”,transforma.paraHTML(“\*\*vinicius\*\*”));

}

Ao executar os testes o framework acusa um erro de compilação do sistema avisando que não existe a classe TransformaTag,

img

Resolvido o problema da ausência da Classe; o teste é executado novamente e mais um problema de compilação ocorre. O método ainda não existe, e mais uma vez criamos um método que sirva somente para consertar este erro, e por fim simplesmente retornamos o null

img codigo

* + 1. Implementação do código

O processo de implementação deve ser o mais simples possível para atender o menor caso possível.

Img

No exemplo conseguimos fazer o primeiro teste passar porém ainda não é um teste com um código funcional e que exerce algum valor ao projeto, para isso vamos criar um novo teste que falhe e que possa nos dar um novo caminho para o código, guiando a algo mais usual e mais reaproveitável. Para isso devemos primeiro criar um novo teste que falhe e nos guie até implementação mais adequada.

@Test

public void transformaUnderlineEmTagU(){

TransformaTag transforma = new TransformaTag();

assertEquals(“<u>vinicius</u>”transforma.paraHTML(“\_\_vinicius\_\_”));

}

E assim podemos criar um código simples e funcional que atenda os dois casos de teste.

código com if

Para concluir a primeira etapa podemos nos satisfazer com o código criado e partir para uma nova bateria de testes que garanta algumas funcionalidades antes de refatorar e remover as duplicações, porém é necessário sempre iniciar com o teste falhando o processo de codificação.

* + 1. Refatoração

Durante a refatoração é importante rodar todos os testes do sistema independente da funcionalidade ou Classe que está sendo desenvolvida, para garantir que tudo está funcionando como antes; caso algum teste falhe é prioridade consertá-lo.

A refatoração mais adequada que existe no código a ser feita é a remoção da grande quantidade de ifs que foram criados para atender o método.

O ideal é utilizar as técnicas de Orientação a Objetos para remover tal duplicação, entretanto para o código exemplificado é fácil resolver utilizando a própria API String contida no núcleo da linguajem Java.

// Codigo

* + 1. Modelo de UML

O TDD deixa o modelo de dados mais desacoplado e testável, assim é possível modificar o designs do sistema de forma segura e com a garantia de que o sistema mantenha seu comportamento quando o processo de refatoração estiver concluído

A cada item que foi concluído da lista, foi aplicado um processo de refatoração focado no design e na remoção da duplicação de código.

//img do degn de classes

A divisão dos pacotes também foram refatorados

//img pacotes

1. Resultados Obtidos

Texto aqui

1. Bibliografia

BECK, Kent. **TDD: Desenvolvimento Guiado Por Testes**. Porto Alegre: Bookman, 2010. 240 p.

CRISPIN, Lisa. **Driving Software Quality**: How Test-Driven Development Impacts Software Quality. In: QUALITY TIME. Quality time. Ieee, 2006. v. 23, p. 70 - 71.

MASSOL, Vincent; HUSTED, Ted. **Junit em ação**. Rio de Janeiro: Ciência Moderna, 2005. 404 p.

WEMBERG, Gerald M.. **Software com qualidade**: Pensando e idealizando sistemas. São Paulo: Makron Books, 1993. 387 p.

KOSCIANSKI, André; KOSCIANSKI, André. Qualidade de Software: Aprenda as metoodologias e técnicas mais modernas para o desenvolvimento de doftware. 2. ed. São Paulo: Novatec, 2007. 395 p.

FEATHERS, Michael C.. Trabalho eficaz com codigo legado. Porto Alegre: Bookman, 2013. 406 p.

BROOKS JUNIOR, Frederick Phillips. O Mitico Homem-Mes: Ensaios sobre engenharia de software. Rio de Janeiro: Campus, 2009. 300 p.

MARTIN, Robert C. et al. Código Limpo: habilidade prática do Agile Software. Rio de Janeiro: Alta Books, 2011. 456 p.

CAELUM (São Paulo). Caelum Ensino e Inovação. Práticas Ágeis de Desenvolvimento de Software em Java: Curso PM-87. São Paulo, 2014.

FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code. Mishawaka: Better World Books, 2007. 431 p.

CUNNIGHAM, Ward. Technical Debt. 2014. Disponível em: <http://c2.com/cgi/wiki?TechnicalDebt>. Acesso em: 22 mar. 2015.

MCCABE, Thomas J.. A Complexity Measure. Ieee: Transactions on software engineeting, Fort George G. Meade, v. 4, n. 2, p.308-319, dez. 1976.

ANICH, Mauricio. Test Driven Development: Teste e Design no Mundo Real. Internet: Casa do Codigo, 2014. 181 p.

DEVMEDIA. Sequência de Fibonacci em Java. Disponível em: <http://www.devmedia.com.br/sequencia-de-fibonacci-em-java/23620>. Acesso em: 22 mar. 2015.

WIKIPEDIA. Refatoração. Disponível em: <http://pt.wikipedia.org/wiki/Refatoração>. Acesso em: 22 mar. 2015.

SIEGMANN, Paul. Download. Disponível em: <http://www.siegmann.nl/epublib/download>. Acesso em: 22 mar. 2015.

FOWLER, Martin. TechnicalDebt. 2003. Disponível em: <http://martinfowler.com/bliki/TechnicalDebt.html>. Acesso em: 22 mar. 2015.

GOMES, André Faria. Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio. Internet: Casa do Codigo, 2014. 176 p.